Acme as an Interactive Translation Environment

Eric Nichols and Yuji Matsumoto
{eric-n,matsu} —at— is.naist.jp

Computational Linguistics Laboratory
Nara Institute of Science and Technology

ABSTRACT

Translation is challenging and repetitive work. Computer-aided translation environments attempt to ease the monotony by automating many tasks for human translators, however, it is difficult to design user interfaces that are easy to use but that can also be adapted to a dynamic workflow. This is often a result of a lack of connection between the interface and the tasks that the user wants to carry out. A better correspondence between task and interface can be achieved by simplifying how software tools are named. One way of accomplishing this to embrace text as the interface. By providing a simple and consistent semantics for interpreting text as commands, the Acme text editor [1] makes it possible to build a system with a text-centered interface. In this paper we explore the implications this has for translation aid software.

1. Motivation

Translation is an essential and important part of human communication. However, it is a very challenging task, requiring the translator to have a complete understanding of and a deep familiarity with the source and target languages. This difficulty is not eased by the fact that fundamentally it is an inherently repetitive task, consisting of looking up unfamiliar words and doing large amounts of editing to produce a good translation.

Given these demands, computers provide a good way to ease this repetitiveness by automating lookup and editing; by converting resources like dictionaries, other translations, and collections of example sentences to a computer-readable format, lookups can be performed much faster. Likewise, sophisticated editing languages can reduce the time and complexity of editing translation results.

The current state-of-the-art for translation aid software, represented by programs like SDL Trados [2] and Wordfast [3], is an environment similar to a word-processor that is centered around providing auto-completion of translation candidates based on a user's past translations. The typical workflow in environments like SDL Trados is as follows:

1. The source document is stripped of its formatting
2. It is split into "translation segments" — short multi-word sequences that are considered easier to translate
3. The user translates each segment sequentially
4. The user carries out any post-editing necessary to produce an acceptable translation

5. The document is automatically cleaned up, removing remaining traces of source text

6. The document's original formatting is reapplied

The interface for programs like Trados may suffice when a user is strictly adhering to this workflow, translating segments sequentially with little pause for lookup or editing, however if users deviate from this usage pattern, it becomes difficult to perform other tasks, such as consulting background material, looking up unknown vocabulary, or performing complex edits. The commands are hard to figure out and remember.

In order to make it easier for the user to figure out and remember how to do something, the interfaces of the tools provided need to closely correspond with what the user wants to do. To make the UI correspond, we need to make it easier to identify commands in a consistent manner.

2. Accessing Our Tools

Consider some conventional ways of identifying commands. One model of accessing tools is to use keyboard shortcuts. Keyboard shortcuts can be used to manipulate individual applications or to call system commands. Keyboard shortcuts can be fast and easy to remember for the right key bindings, but fundamentally, they are an obscure way of identifying commands. As the number of shortcuts increases, so does the burden on the user's memory. Also, the small number of easy to type shortcuts makes keyboard shortcuts unsuitable for programs that have a large number of actions. Emacs exemplifies this problem having become so complex that it requires multiple modifier keys to be held while typing sequences in order to access some functionality.

An alternative to keyboard shortcuts is the use of menus. While clicking a menu entry may be easier than using keyboard shortcuts, hierarchical menus still suffer from the same obfuscation problems that plague keyboard shortcuts, only in the case of menus, users end up classifying their commands in a hierarchy. Another problem with menus is that they are static in nature; updating the menu layout can confuse users, and, in some implementations require alteration of the underlying program. This makes menus too difficult to adapt to changing tasks or workflows.

Where is the consistency in these approaches? Operating systems often provide human interface guidelines that are intended to provide consistent keyboard shortcuts and menu layouts across applications. Examples of this are the Command+N and Command+W shortcuts in Mac OS X for creating and destroying windows. But ultimately, this consistency is difficult to enforce; third party applications often break the rules, and operating systems, like Linux, that lack enforcement authorities are unable to maintain a high level of consistency.

One option for identifying commands that we have not yet considered is the plain text interface. With the advent of GUIs, plain text interfaces are often dismissed as not being user friendly enough. This bad reputation of the command line is due to the perceived obscurity of the names of some commands and the inflexible keyboard-only nature of older terminals.

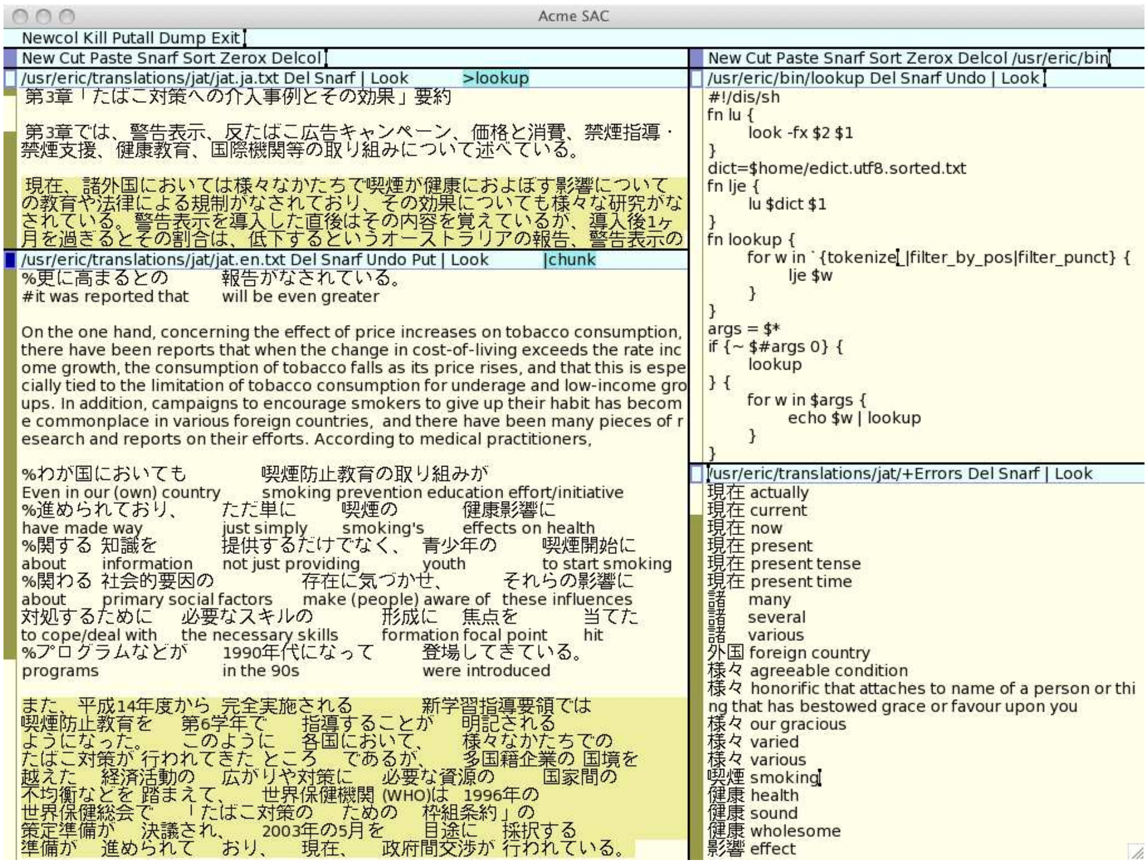


Figure 1 *Translating a Japanese document into English using Acme.* The upper-left buffer contains the source document. Middle-clicking on the text `>lookup` that is highlighted in aqua in the title bar of the upper-left buffer looks up the English translation of each word in the text highlighted below. The results of the lookup are displayed in the lower-right buffer. `lookup` is a shell function defined in the upper-right buffer. The lower-left buffer contains a translation in progress. At the top of the buffer is the finished English translation of a Japanese paragraph. Below that, a paragraph has been divided into tab-delimited translation units, and a rough English translation has been given below. At the bottom of the buffer is a Japanese paragraph that has been divided into translation units by the `|chunk` command highlighted in the title bar of the lower-right buffer.

3. Acme for Poets

The Acme text editor and programming environment makes it worth reconsidering plain text as an interface. In Acme the mouse is used to provide a consistent semantics for treating text as commands. This echoes the lack of distinction between a program and data that is common in functional programming languages like Lisp and Scheme. In terms of user interfaces, this frees the users from having to remember obscure shortcuts or navigate complicated menus to access their tools; all they need to do is type the name of the command. This reduces the burden on the user's memory and makes it much simpler to carry out the desired action: simply click on the text of a command to execute it. Acme takes care of interpreting the text as a command by sending a request to the shell where the requested command is executed and the output is displayed in a new buffer. An example of Acme in action is given in Figure 1.

This design has some interesting implications. Because any text can be executed as a command, the user is not limited in terms of what tools he or she can access, keeping the workflow dynamic and allowing it to easily adapt to the needs of a given task. At the same time, the clear division in roles — the keyboard for text production, and the mouse for text interpretation — as well as the clean semantics of the mouse buttons for select/execute/get provides consistency to the UI. By providing an easy way for text to be interpreted as commands, Acme combines the command and its interface, removing the disconnect between what the user wants to do and how it is done. This makes the UI easier to figure out and remember, and produces an interface that is more suitable for text-based tasks.

Acme makes applying tools to data as easy as clicking on text with a mouse. Furthermore, the small tools design philosophy that inspired Acme makes it easy to apply existing NLP tools in Acme. This means that tasks like using reference materials or editing text are easy. Users who frequently deal with language, such as linguists, translators, language learners, or poets need to be able to efficiently explore and manipulate language. Acme can provide a powerful working environment for users who spend a lot of time interacting with text.

Acme is clearly attractive for text-heavy tasks like translation, however, some work needs to be done to produce a usable environment. Tools and intuitive interfaces need to be constructed for common text manipulation tasks such as splitting a sentence into words or phrases, looking up words in dictionaries, or translating phrases using translation memories or machine translation systems. Support for multi-lingual input needs to be improved: we need redistributable fonts with good Unicode coverage and good input method editors for handling input and display of text in languages that do not use Latin-based scripts. Finally, we need a way of getting Acme to the end user in a simple, easy to install package that works out-of-box.

4. Consistent Text Interfaces for Multi-lingual NLP Tools

The advent of Unicode and the rewrite of the UNIX toolchain in the Plan 9 [4] and Inferno [5] operating systems with native UTF8 support have made the small tools approach viable for multi-lingual NLP tools. Here we will show how a simple program for language identification can combined with shell functions to provide a consistent language-independent interface under Inferno for a few common NLP tools that come in handy during translation. We use Limbo and sh in our examples, but these techniques are equally as applicable to C and rc on Plan 9.

Consider the task of tokenizing a sentence into words. Many NLP tools, such as the

dictionary search function, `lookup` in Figure 1, operate on word-level information, so this task is an essential form of preprocessing for applying them. We will build an interface for the languages English and Japanese since the actual method of tokenization is very different and illustrates the need for a simple, consistent interface.

The shell function `tok_en` tokenizes an English sentence by splitting on any punctuation and removing extraneous whitespace. While this approach is sometimes over aggressive in splitting — hyphenated words and words with apostrophes will occasionally be segmented unnecessarily — it is a simple, easy to implement heuristic.

```
fn tok_en {
    sed 's/([!"#$%&'()*+,-./:;<=>?@[]^_`{|}~])/ 1 /g
         s/ +/ /g
         s/^ +//g
         s/ +$//g'
}
```

Here is the Japanese equivalent:

```
fn tok_ja {
    tcs -f utf-8 -t euc-jp |
    mecab -Owakati | # suppress POS output, tokenizing into words
    tcs -f euc-jp -t utf-8
}
```

`Mecab` [6] is a Japanese morphological analyzer. It is typically used to tag words in a sentence with part-of-speech information, but, in the same way `wc` can be used to count characters, words, and lines because they are calculated in the same pass, `mecab` can be used to tokenize a Japanese sentence, because word boundaries and parts-of-speech are determined together. The output mode `wakati` indicates that a sentence tokenized with whitespace should be returned without part-of-speech information. Since `mecab` expects EUC-JP encoded input, we use `tcs` to convert the Japanese text to and from Unicode.

We combine `tok_en` and `tok_ja` together with the following shell function:

```
fn tok_any {
    args = $*
    if {~ $1 -e} {                # English
        f = tok_en
        (lang args) = $args
    } {~ $1 -j} {                 # Japanese
        f = tok_ja
        (lang args) = $args
    } {                           # Use English as fallback
        f = tok_en
    }
    $f $args
}
```

where setting a flag determines which language is being processed and calls the appropriate tokenizer. If no flag is set, the English tokenizer is used as a fallback.

`tok_any` acts as a multiplexer, calling the proper language-specific implementation of a given task based on its settings. Consistent naming and pipe based I/O makes this possible. It does not matter how different the implementations of the English and Japanese tokenizers are; they are encapsulated in separate functions, but combined together they represent a language-independent task.

5. Language Identification

Manually indicating the language each time `tok_any` is called is inconvenient. It would be useful to have a way of automatically identifying the input language.

`Whichlang` is a limbo function similar to Plan 9's `freq(1)` that uses a simple character frequency based heuristic to identify the language of an input stream of text. Occurrences of alphabetical characters or punctuation commonly used in English writing are counted as evidence for English, whereas occurrences of characters typically used in Japanese — such as hiragana, katakana, or Han ideographs — are taken as evidence for Japanese. This is a simple approach that could be improved on, but it can easily be expanded and will suffice for our current purposes.

```
whichlang(fd: ref Sys->FD): string
{
    EN: con 0; JA: con 1; FLOOR: con 0.5;
    lang := array[2] of { 0, 0 };
    buf := array[256] of byte;
    c:= 0;
    for(;;) {
        n := sys->read(fd, buf, len buf);
        s := string buf[0:n];
        if(n <= 0)
            break;
        for (i := 0; i < len s; i++) {
            if (s[i] != ' ') {
                c++;
                case s[i] {
                    'a' to 'z' or 'A' to 'Z' or
                    '!' to '/' or ':' to '?' =>
                        lang[EN]++;
                    '�' to '�' or # Asian punctuation
                    '�' to '�' or # hiragana
                    '�' to '�' or '�' to '�' or # katakana
                    '�' to '�' or # kanbun
                    '�' to '�' or # CJK unified ideographs ext.
                    '�' to '�' or # CJK unified ideographs
                    '�' to '�' => # half- and full-width forms
                        lang[JA]++;
                }
            }
        }
    }
    l := "";
    max := 0;
    if ((lang[EN] > max) && ((real lang[EN] / real c) > FLOOR)) {
        l = "en";
        max = lang[EN];
    }
    if ((lang[JA] > max) && ((real lang[JA] / real c) > FLOOR)) {
        l = "ja";
        max = lang[JA];
    }
    return l;
}
```

You may notice that many characters in the case statement in `whichlang` could not be displayed. This is because they are outside of the Unicode range covered by the fonts available in Plan 9 and Inferno. We will return to this problem in Section 8.

Turning whichlang into a Limbo module allows our language detection facilities to be used in any other Limbo program. Writing a wrapper to use whichlang as a stand-alone program is trivial. Assuming such a program, we can write a shell script that will automatically set the language flag for any program to the language of its input.

Setlang takes as its first argument the command whose language is to be set. It determines if it has been called with any manually set language flags, and, in their absence, it uses tee to cache a copy of the program's input and pipe a portion to whichlang for identification. If a language is successfully identified, the language flag is set accordingly. Finally the command is called with the language flag either set or omitted entirely.

```
fn setlang {
    args = $*
    or {~ $#args 1 2} {
        echo 'usage: setlang <command> [-e | -j]' >[1=2]
        raise usage
    }
    (com args) = $*
    (and {~ $#args 1} {~ $args -e -j} {
        (lang nil) = $args
    })
    tmp = ${pid}^.tmp
    if {~ $#lang 0} {
        l = `{tee $tmp | sed 200q | whichlang}
        if {~ $wl en} {
            lang = -e
        } {~ $l ja} {
            lang = -j
        }
    }
    if {ftest -f $tmp} {
        $com $lang < $tmp
        rm -f $tmp
    } {
        $com $lang
    }
}
```

With setlang we can complete our tokenization interface as follows:

```
fn tokenize {
    setlang tok_any $*
}
```

Combining setlang with a flag-based multiplexer like tok_any is a powerful abstraction. The user is presented with a single, simple command that works regardless of the language of the input and does not require any further intervention. We have used setlang and the design pattern introduced with tok_any to create interfaces for several common tasks including dictionary lookup, part-of-speech tagging and dependency parsing. Figure 1 contains an example of a dictionary lookup interface that uses the tokenize function. As we can see from this example, these interfaces make useful building blocks in shell scripts, but they also extend Acme's interface in meaningful ways.

The examples of whichlang and setlang show how easy it is to use the Plan 9 design philosophy and the tools provided by Inferno to create simple, consistent text-based interfaces for multi-lingual NLP applications. Clearly, Inferno is an attractive

platform for NLP development, however, the addition of Acme also makes it ideal for the distribution of NLP services. What is needed is an easy way to get Acme and useful NLP tools into the hands of the end user.

6. Providing a Translation Environment with Acme SAC

The Acme Stand Alone Complex [7] project satisfies many of our requirements. Acme SAC is a slimmed-down distribution of the Inferno operating system designed to present Acme as a single application. It simplifies the user experience by focusing on Acme as the interface rather than presenting a full window manager with individual applications as in the full Inferno distribution. Because Inferno runs in an emulator on top of a virtual machine, Acme SAC can run on many operating systems without any external dependencies. Acme SAC's designer has focused mainly on the Windows version, but a Linux version is also available.

Acme SAC simplifies the user experience further by eliminating any installation or setup process. Users simply download a tarball and run a single application to start Acme. The first time Acme is started, a new user account with reasonable default settings is automatically created. Inferno is capable of accessing the local file system, and Acme SAC mounts and makes it available by default. The os command provides access to host OS commands, allowing users to continue using any of their existing tools that support pipe-based I/O, and Acme SAC clients can communicate with Inferno installations, making it easy to run commands on remote machines.

To help users get accustomed to the Acme environment, Acme SAC is distributed with a README that acts as an interactive tutorial thanks to Acme's dynamic text interpretation capabilities. Users learn how to use the mouse to select and execute text, how to browse man pages, and how to use Acme SAC's IRC client. Every text example is immediately available for exploration to the user.

Our goal is to provide a set of tools that will help our target users group of linguiphiles work with text. The classic text "Unix(TM) for Poets" [8] by Ken Church can act as an interactive README for NLP, however, its examples need to be updated for the Inferno shell, and perhaps, to include text in languages other than English. We are in the process of updating it (the new text will be called "Acme for Poets"), however, the lack of `awk` and `paste` commands make some of the examples challenging.

7. Acme SAC for Mac OS X

One of our goals is to make Acme available as a translation environment to as many potential users as possible. To do this, it is important that Acme SAC run on as many platforms as possible and support I/O in as many languages as possible. Thus, we decided to package Acme SAC for Mac OS X. Since an Inferno port already exists, packaging was largely a matter of adding the necessary Mac OS X specific files from Inferno and resolving changes in code that had occurred after Acme SAC branched. There were some interesting challenges in packaging Acme SAC for Mac OS X and handling multilingual I/O that we will now describe.

Mac OS X applications are typically distributed in a single directory that contains all of the application's dependencies. This makes it easy for users to manage; installation and upgrades consist of simply dragging the application to a desired folder, however, this caused problems for some of Acme SAC's settings. For example, by default, Acme SAC creates its user directories inside of the Acme SAC tree. This is problematic for Mac OS X because a subsequent install would overwrite all of the user's data, and it causes

security issues when installed by an administrator. Moving the acme home directory into the Mac OS X user's home directory seemed most appropriate, however, it would be inappropriate to bind `/Users/me` directly to Acme SAC's user directory since that could cause security problems. The solution was to create a special directory in the Mac OS X user's home directory called `acme-home/` and copy all Acme SAC users files there during new user creation. This protects the user's other files in the event that something goes wrong with Acme SAC. In addition, we bind Acme SAC's `/tmp` onto `acme-home/` as well, making Acme SAC's root directory truly read-only.

7.1. Multi-lingual I/O

Input Method Editors and high-coverage, redistributable Unicode fonts are absolutely essential to our goal of using Acme SAC to provide a translation environment capable of supporting a variety of languages.

While Plan 9 and Inferno both support limited Unicode character input via modifier keys, languages that do not use Latin-based scripts often require more sophisticated forms of managing input. There are native IMEs, such as `ktrans`[9] , that make input conversion possible in Plan 9 and Inferno, but they are often limited in the scope of languages covered. In contrast, Mac OS X and Windows both provide IMEs capable of handing a large number of languages, and can dedicate more resources to improving the quality of input conversion than a lot of smaller projects can.

For our purposes, the IMEs of Acme SAC's host OSes also provide the user with consistency; he or she is not required to learn a new method of entering text to do translation in Acme SAC. Acme SAC's graphical component is implemented using it's host operating system's native window system API, making it easy to provide IME support in a minimal amount of code in a manner that does not disturb users who do not need the IMEs. In fact, when we questioned Acme SAC's creator on how the Windows version's IME support was implemented, he was not aware of its implementation at all (personal correspondence). We were able to implement support for the IME in Mac OS X with only a few dozen lines of code — the majority updating the the Mac encoding of special keys to their Unicode equivalents.

8. Discussion

With the enhancements to multi-lingual processing we have made by creating consistent, language-independent interfaces using `whichlang` and `setlang` and by providing IME support, Acme SAC has made a lot of progress toward becoming a viable translation environment. However, there are still some issues we need to consider.

One problem that arises with increased support of languages is that fonts are required to display them. This problem is well-illustrated in the case statement in `whichlang` from Section 5*. There are often legal issues in the redistribution of fonts — the fonts remain one of the few parts of Inferno that has not been open sourced. Currently, Acme SAC is distributed with a font that only covers the Latin alphabet and a small number of special Unicode symbols. This is impractical for a translation environment. We must find Unicode fonts that cover the languages we want to translate and include them in

* It may be of some consolation to Peter Weinberger's detached and pixelated face that the characters in the Unicode class boundary tests are often very rare, and not many native speakers of Japanese ever use them, but this does not diminish the need for distributable Unicode fonts of high quality and coverage.

Acme SAC. We have not yet conducted a thorough investigation of the fonts available for Plan 9, but perhaps approaches like `ipa_install` [10], a scripts that automatically downloads Japanese fonts in a way that satisfies their license and converts them for use on Plan 9 and Inferno is an useful approach to help solve the font distribution problem.

One of the reasons for the widespread adoption of SDL Trados is its ability to let the user translate popular document formats like Microsoft Word and Adobe Pagemaker, stripping away the formatting before translation and replacing it afterward. In order to attract users, it may be necessary to handle translation of such documents as well. The raises the more general issue of how (if at all) markup should be handled in Acme. Programs like `strings(1)` make it possible to extract plain text from Word documents, but would it be possible to read and write in the Word format without losing formatting information?

Another asset of state-of-the-art translation aid environments is their support of translation memories and auto-complete. This starts with dividing the input text into smaller translation units. Such "text chunking," as it is known in the field of natural language processing, is easy to replicate with existing tools. Indeed it was one of the first features we implemented in Acme. With the text to translated divided into chunks, we need a fast way of navigating these chunks. Acme SAC's client wrapper for Charon may provide some guidance: adding a special client with Prev and Next buttons to move from chunk to chunk could ease navigation of the document being translated. We will also need to provide text alignment facilities in order to construct translation memories from document pairs. Investigating an appropriate way of implementing this in Acme and Inferno remains an area of future work.

Finally, perhaps the most important question concerns Acme itself. While it is our position that Acme's text based interface simplifies interaction with text by eliminating the need for nested menus and keyboard shortcuts, we have not empirically investigated Acme's learning curve. Thus, answering the question of whether or not one has to become an Acme power user in order to translate using it is of utmost importance.

9. Conclusion

Acme's text-driven nature removes the disconnect between tools and their interfaces. This makes figuring out how to do things more intuitive to the user and reduces the burden on his or her memory. This improves the interface and, in doing so, makes the user's job easier. Acme SAC is an ideal means of packaging the Acme environment for the user. It runs all of the major operating system, requires no setup on the part of the user, and provides seamless access to tools and data on remote machines. Inferno makes it easy to provide consistent, multi-lingual text interfaces for completing common NLP tasks, and it shows promise as a platform for multi-lingual platform for text processing.. As a first step, we have packaged Acme SAC for Mac OS X and implemented support for its native IME. We also outlined some of the challenges that will have to be overcome in order to fully realize Acme as an environment for interactive text manipulation.

10. Acknowledgements

We would like to thank Noah Evans and Uriel Archangel for their feedback and encouragement, Francis Bond for his valuable perspective as an NLP researcher, Caerwyn Jones for creating the program that made this research possible and for the guidance in

getting Acme SAC to compile under Mac OS X, and the many users who provided testing and feedback on Acme SAC for Mac OS X. Eric Nichols would also like to thank the Ministry of Education, Culture, Sports, Science, and Technology of Japan for financial supporting his research.

11. References

[1] R. Pike. *Acme: A User Interface for Programmers.* Proceedings of the USENIX Winter 1994 Technical Conference, 1994, pp. 223–234.

[2] SDL Trados Technologies. *SDL Trados.* `http://www.trados.com/en/`

[3] Wordfast LLC. *Wordfast.* `http://www.wordfast.net`

[4] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, P. Winterbottom. *Plan 9 from Bell Labs.* Computing Systems, 38(3), Summer 1995, pp. 221–254.

[5] R. Pike, S. Dorward, et al. *The inferno operating system.* Bell Labs Technical Journal, 2(1), Winter 1997, pp. 5–18.

[6] T. Kudo. *MeCab: Yet Another Part–of–Speech and Morphological Analyzer.* `http://mecab.sourceforge.net/`

[7] C. Jones. *Acme Stand Alone Complex.* `http://code.google.com/p/acme-sac`

[8] K. W. Church. *Unix(TM) for Poets.* Unpublished manuscript.

[9] K. Okamoto. *Ktrans.* `http://basalt.cias.osakafu-u.ac.jp/plan9/s11.html`

[10] Tokyo Inferno Plan 9 User Group. *TIP9UG JAPANESE FONT PROJECT.* `http://www.tip9ug.jp/wiki/jp_fonts/`